History Viewer: Displaying User Interaction History in Visual Analytics Applications

Vinícius C.V.B. Segura[1,2(✉)] and Simone D.J. Barbosa[1]

[1] Departamento de Informática, PUC-Rio, Rio de Janeiro, Brazil
{vsegura,simone}@inf.puc-rio.br
[2] IBM Research, Rio de Janeiro, Brazil
vboas@br.ibm.com

Abstract. Effective and efficient strategies are needed to extract unknown and unexpected information from data of unprecedentedly large size, high dimensionality, and complexity [7]. Only a combination of data analysis and visualization techniques can handle these complex and dynamic data [4]. Visual analytics applications aim to integrate the best of both sides.

After the knowledge discovery process, a major challenge is to filter the essential information that led to a discovery and to communicate the findings to other people. We propose taking advantage of the trace left by the exploratory data analysis, in the form of user interaction history. This paper presents a framework to instrument web visual analytics applications, logging the user interaction during the exploratory data analysis. This paper also presents our solution to display the user interaction history to the user, enabling him to revisit the steps that led to an insight.

Keywords: Visual analytics · User interaction logging · User interaction history · Data visualization · History visualization · Log visualization

1 Introduction

We are now living in a Big Data world. We generate 2.5 quintillion bytes (2.5 10^{18} bytes or 2.5 exabytes) of data every day, meaning that 90 % of the the available data today has been created in the past two years.[1] Research-wise, the bottleneck has shifted from *data acquisition* (when there are poor datasets) to *data analysis* (what to do with the rich datasets recently available) [5].

Human attention is now a limiting resource. Effective and efficient strategies are needed to extract unknown and unexpected information from these data of unprecedentedly large size, high dimensionality, and complexity [7]. Only a combination of data analysis and visualization techniques can handle these complex and dynamic data [4].

[1] IBM – What is big data?, available at: http://www-01.ibm.com/software/data/bigdata/what-is-big-data.html.

On the one hand, computers can provide intelligent data analysis [6] without cognitive biases [3]. Their enormous processing power [1] and superior working memory [3] guarantee an incomparable mathematical, algebraic, and statistical prowess to handle massive volumes of data. On the other hand, human users can contribute with their analytical capabilities and inherent visual perception [6], which enable them to perform visual information exploration [1].

Visual analytics applications (VAApps) aim to integrate the best of both worlds. They pre-process and analyze data, presenting it to users in a way to take advantage of the user's cognitive capabilities. The final outcome may be an unexpected insight – one only possible by combining computation and cognition.

After the knowledge discovery process, a major challenge is to filter the essential information that led to a discovery and to communicate the findings to other people. We propose taking advantage of the trace left by the exploratory data analysis, in the form of user interaction history. We have devised a framework to instrument web VAApps to log the user interaction during the exploratory data analysis. This user interaction history can be later presented to the user, enabling him to revisit the steps that led to an insight.

In this paper, we start by presenting the framework and its different components (Sect. 2). In the following section (Sect. 3), we discuss the associated log model, which makes the bridge between the VAApp and our history viewer. In Sect. 4 we detail our history visualization idea, elements, and concepts. Next, (Sect. 5) we show how our solution was applied to a weather insights VAApp. Finally, we conclude this paper (Sect. 6) with some final remarks, discussing the current solution limitations and proposing some future work.

2 Framework

Figure 1 summarizes the different components of our framework and the relations between them. We consider that the VAApp architecture is itself comprised of three basic components:

1. A **web UI** – the front-end – responsible for rendering the VAApp in the user's browser. This usually corresponds to the HTML and JavaScript code to be executed in the client's browser.
2. A **data service** – the back-end –, which communicates with the database(s) and sends data back to the web UI according to a given API. The technology used in this component can be various: Node.JS, Java, Python, etc. It is important to notice that this component may not be deployed in the same application as the front-end, or even be developed by the same VAApp team. For example, the VAApp may use a 3rd party API and make transformations to the data only on the client side.
3. Finally, the **data** itself, which is usually stored in some sort of database. Again, there is a multitude of technologies that can be used for this component – from local stored files to cloud-based storage –, which are beyond the scope of this discussion.

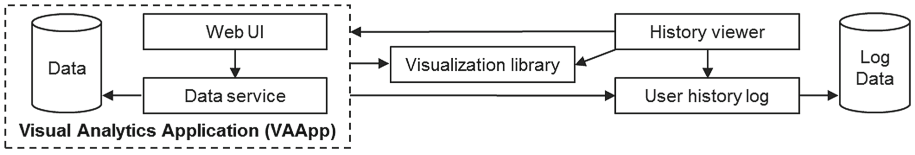


Fig. 1. Framework components diagram.

Our framework considers this basic VAApp architecture, and builds upon it with three components:

1. A **visualization library**, providing reusable basic charts to be used in different VAApps. This component uses d3.js[2] and both the VAApp and the history viewer may use it.
2. An **user history log**, storing the user interaction history. Similar to the VAApp's `data service`, this component handles the queries to the `log data` stored in some sort of database. Section 3 details our current log model.
3. A **history viewer** component, a VAApp itself that presents the user interaction history in a comprehensible way. It corresponds to the VAApp's `web UI`. We will present our visualization in Sect. 4.

3 Log Model

In order to visualize what is being logged by the VAApp, we must define a common log model to be shared between the VAApp and the history viewer. We propose the model illustrated in Fig. 2. It is a hierarchical model, which can be split in two: the definitions and the interactions hierarchy.

On the right-hand side, we can see the **definitions hierarchy**. This hierarchy is established at development time by the VAApp developer. Its root is the **VAApp definition**, which has a collection of `view` and `data service definitions`. A **view definition** represents a view (page) that can be navigated inside the VAApp and contains a collection of `visualization component definitions`. A **visualization component definition** represents a visual element (*e.g.*: a chart, a map, a collection of charts) that appears on the UI. The **data service definition** represents the data service APIs used by the VAApp to gather data for its visualizations.

These definitions consists mostly of a given name, a version, and a URL. The versioning is managed by the VAApp developer. He is therefore responsible for maintaining retro-compatibility (by providing different URL's for different versions, for example). Some definitions (`data service` and `visualization component definitions` – the leaves of the hierarchy) must also provide a class name, so they can be later reused.

[2] http://d3js.org/.

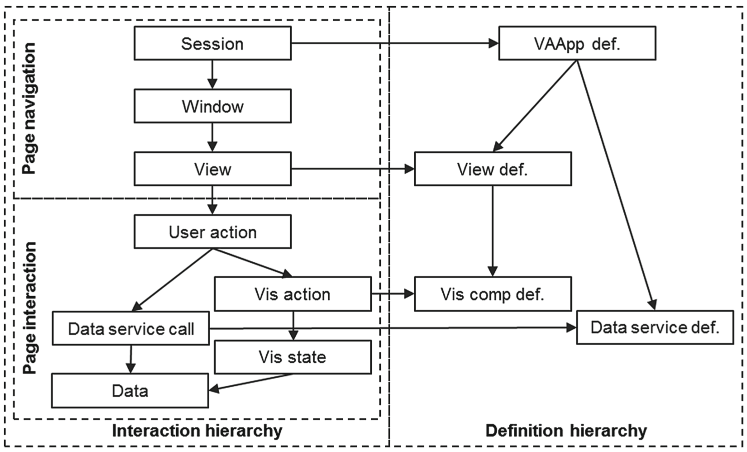


Fig. 2. The log model.

If we take the Gapminder[3] website as an example, the `VAApp definition` would point to the website (http://www.gapminder.org/). For visualization purposes, we could say that there would be only a single `view definition`, pointing to the Gapminder World page (http://www.gapminder.org/world/), ignoring the other pages (video, download, etc.). This page contains a single scatterplot, so it would only have one `visualization component definition`. To keep things simple, we could say that there is only one data service API which returns a time-series data given an indicator. This API would be the only `data service definition` of this example VAApp.

On the left-hand side, we can see the **interactions hierarchy**. It starts with a **session**, when the user logs in to a VAApp. It keeps a reference to the `VAApp definition` and may contain additional information, such as the browser's user agent, for example. During a `session`, the user can open different browser **windows** (or tabs). In each `window`, the user may navigate through many VAApp `views`, each one of which referencing the corresponding `view definition`.

The steps described so far are more related to the page navigation in the browser than an actual user interaction within the page. The first **user action** usually corresponds to the actual page loading and the visualization components' initial state. A user action may trigger a **data service call** to gather new `data` from the `data service` component. Since we focus on changes in the visualization components, a **user action** should always have at least one `visualization action`, indicating what type of change happened to the `visualization component`. Every `visualization action` generates a new **visualization state**, describing the state of the `visualization component` after the action and referencing the `data` currently represented. As the user interacts with the `view`, new `user actions` are recorded every time a `visualization component` is changed.

[3] http://www.gapminder.org/.

We believe that the VAApp developer may give some semantics and context to the `user action`, but cannot infer the user's intentions or higher-level goals. We may see this limitation as similar to the affordance levels [9]. At the operational level, we have individual actions (mapped onto `visualization actions`). At the tactical level, we find a sequence of actions that were executed to achieve goals and sub-goals (mapped onto `user actions`). Finally, we cannot log the strategic level, since it relates to the conceptualizations regarding problem formulation and problem solving processes.

The `user actions` therefore can be described by the VAApp developer using terms related to the VAApp instead of more generic ones. For example, in Gapminder, we could describe the change of the y-axis as "Changed y-axis indicator to ACME" instead of generically saying "Changed y to ACME".

We believe that more specific information can make it easier for the user reading the history log to recall the actual actions that took place, since it will be a more contextual description that reflects the actual VAApp UI he first interacted with.

The `visualization actions` are categorized according to a given set of **tasks**, so the VAApp developer does not need to provide any additional information. We are using Brehmer's and Munzner's [2] multi-level typology of abstract visualization tasks, summarized in Fig. 3, to categorize the `visualization actions`. We focused on the "how" part – "families of related visual encoding and interaction techniques" – since the "why" part falls onto the strategical abstraction level previously discussed and is, therefore, outside the scope of this work. The available `visualization actions` are, therefore:

- **Encode**: Codify data in the visual representation.
- **Select**: Demarcate one or more elements in the visualization, differentiating selected from unselected elements (*e.g.*: select, brush, highlight).
- **Navigate**: Alter user's viewpoint (*e.g.*: zooming, panning, rotating).
- **Arrange**: Organize visual elements (*e.g.*: reordering axes, rows/columns).
- **Change**: Alter visual encoding (*e.g.*: size and transparency of points, changing the chart type).
- **Filter**: Adjust the exclusion and inclusion criteria for elements in the visualization.
- **Aggregate**: Change the granularity of visualization elements.
- **Annotate**: Add graphical or textual annotations associated with one or more visualization elements.
- **Import**: Add new elements to the visualization.
- **Derive**: Compute new data elements given existing data elements.
- **Record**: Save or capture visualization elements as persistent artifacts.

By using this typology, we believe it will be possible to compare visualization actions from different interaction paths and even from different VAApps. Moreover, we believe it will also simplify their interpretation, since we limit the number of possible tasks.

Continuing with the Gapminder example, when the user navigates to the Gapminder World page, all the `page navigation` hierarchy is constructed, with

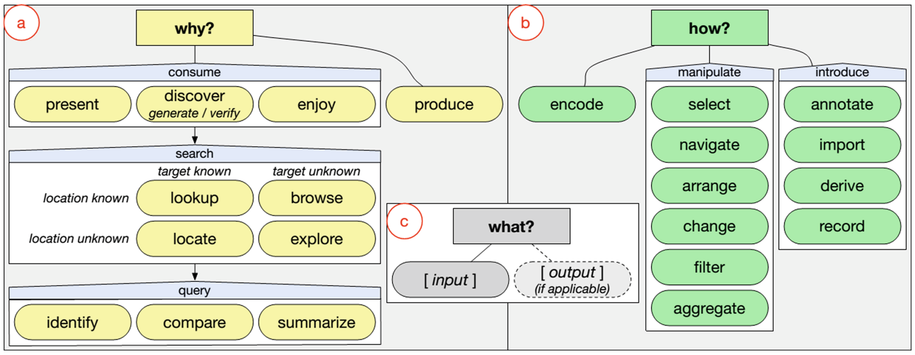


Fig. 3. Brehmer's and Munzner's multi-level typology of abstract visualization tasks.

the `view` referring to the Gapminder World `view definition`. The page loading is mapped to an `user action`, which (considering our simple data API), would trigger three `data service calls` (to gather data for each axis and the circles' sizes using the `data service definition`) and one `visualization action` (the `encoding` of the scatterplot `visualization component`).

If the user hovers the mouse pointer over a data point, a new user action would be created with just a `visualization action` (the `annotation` of the data point), since no new data seems to be gathered. If the user changes an axis indicator, again a new `user action` is created, which would have a single `data service call` (data for only one axis) and a `visualization action` (`encoding` new data on the scatterplot).

4 History Visualization

After logging the user history, we want to present it back to the user so he can trace back the steps in his interaction. We propose the representation illustrated in Fig. 4. We based our visualization in the GIT commit graph,[4] so users could have some sense of familiarity (at least amongst GIT users). We chose to keep the y-axis as a time axis anchor, so in the "worst-case scenario" the user may just read through the description texts from top to bottom and still relate to the interaction he had with the VAApp.

Each column in the representation groups different `views` if there is no time conflict, *i.e.* the time span of the `view` (time between the first and last `user actions`) does not overlap the time span of another `view`. From the figure, for example, we can notice two parallel `views` (two columns), so we can infer that the user had at least two open `windows`.

`View` navigations are represented as column-wide ellipses breaking the flow. For example, looking at the fourth row, second column, we see that the user navigated from one view to another, given the break represented by the ellipse.

[4] http://chimera.labs.oreilly.com/books/1230000000561/ch01.html#fig0101.

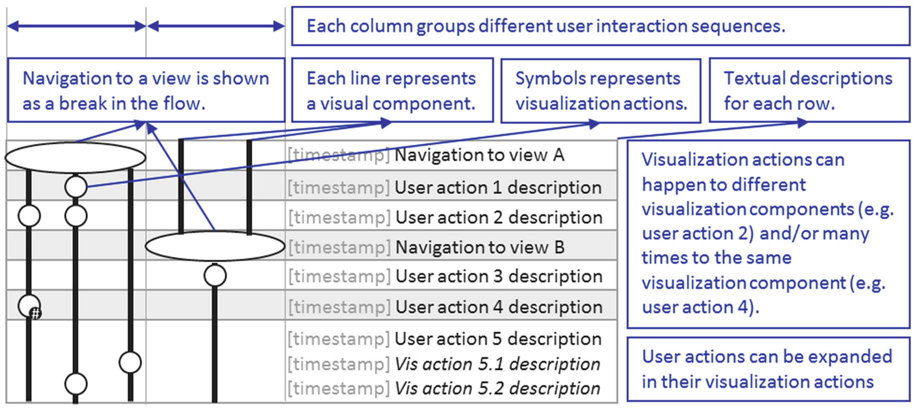

Each column groups different user interaction sequences.
Navigation to a view is shown as a break in the flow.
Each line represents a visual component.
Symbols represents visualization actions.
Textual descriptions for each row.
[timestamp] Navigation to view A
[timestamp] User action 1 description
[timestamp] User action 2 description
[timestamp] Navigation to view B
[timestamp] User action 3 description
[timestamp] User action 4 description
[timestamp] User action 5 description
[timestamp] Vis action 5.1 description
[timestamp] Vis action 5.2 description
#
Visualization actions can happen to different visualization components (e.g. user action 2) and/or many times to the same visualization component (e.g. user action 4).
User actions can be expanded in their visualization actions

Fig. 4. An annotated history visualization. The elements in blue are not part of the representation, but comments included here to help describe it.

From each `view` representation a different number of lines emerge. Each one represents a `visualization component` of that given `view`. In the first column, therefore, we could say that the `view` has three different `visualization components` and, in the second column, the initial `view` had two `visualization components` and navigated to one with a single `visualization component`.

We chose to represent `views` occupying the whole column to highlight breaks in the flow and changes in the number of `visualization components`. If the second column did not begin with those two lines and simply started with the ellipse on the fourth row instead, we could infer that the user had just opened another `window` at that point in time.

In each `visualization component` line we can have multiple symbols, representing the different tasks associated with the `visualization actions`.

The line in which the symbol appears indicates in which `visualization component` the action took place. The order of the lines, therefore, should be consistent amongst different representations of the same `view`, so the user can create a mental mapping of which line is which `visualization component`.

Finally, each banded row represents a single `user action`. A `user action` may group multiple `visualization actions`. Looking at the figure, for example, we can see that the `user action` 2 comprises of two different `visualization actions` for two different `visualization components` (one symbol in each of the first two lines, on the same row), whilst the fourth `user action` has a number of `visualization actions` for the same `visualization component` (represented by the small '#' badge attached to the circle). If the user wants to look at each individual `visualization action`, he can expand the `user action`, as demonstrated with the `user action` 5 in the figure.

5 Early Implementation

For our first implementation, we chose WISE - Weather InSights Environment [8] as our target VAApp. WISE's main UI (shown in Fig. 5) is composed mainly of three visual components: a map in the background, an event profile at the bottom, and meteograms on the right. We disregarded the top card – a configuration card – since it is only a series of common HTML input elements to choose/display the current parameters (forecast, grid, property, and timestep) and we wanted to focus on the visual analytics aspects.

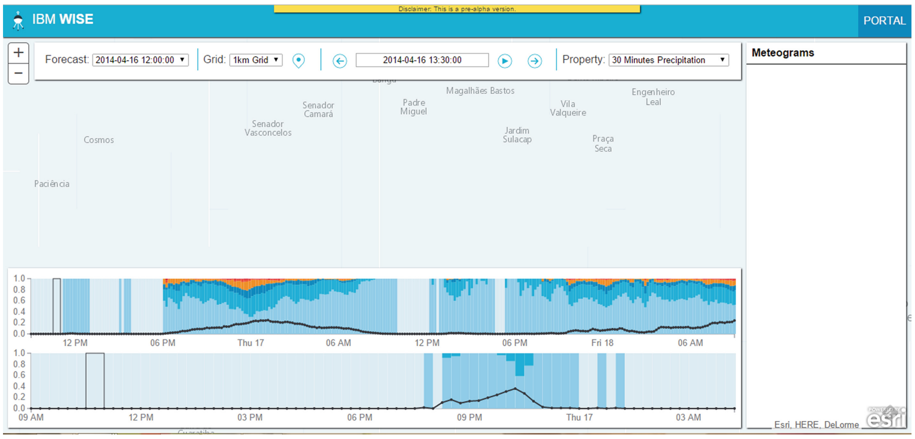

Forecast: 2014-04-16 12:00:00
Grid: 1km Grid
2014-04-16 13:30:00
Property: 30 Minutes Precipitation

Fig. 5. WISE's main UI.

We started by mapping the available interactivity and visualization changes they generate onto our log model, in order to evaluate how well our approach would fit. The results can be seen in Table 1, with the `visualization action` task inside parentheses.

We proceeded to instrument WISE to generate a log according to our model. We developed helper code to reduce the work needed to be done by the VAApp developer and the impact on the client side. Thus, the instrumentation comprises only four main concerns: (i) keeping the **definition hierarchy** updated; (ii) using the auxiliary **logger class**; (iii) using compatible **visualization components**; (iv) using compatible **data services**.

The `definition hierarchy` is expressed in JSON. We created a simple UI to post this JSON data and get the ids corresponding to the created definitions. These ids should be used at run time to make the `interaction hierarchy` reference the `definition hierarchy`.

The logger class helps to post to the `user history log` component. The developer informs when a `session` starts, the current `view`, and the page interactions. The logger manages saving `session` and `windows` ids, and also doing the actual posting.

Table 1. WISE's user actions and visualization actions study.

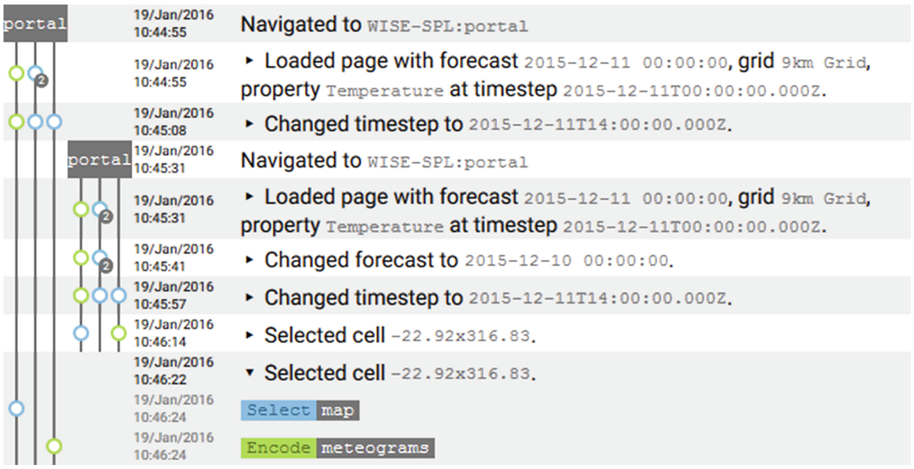
User action	Visualization action
Change in the forecast	Redraw map (`Encode`)
Change in the grid	Redraw profiles (`Encode`)
	Redraw meteograms if necessary (`Encode`)
Change in the property	Redraw map (`Encode`)
Change in the timestep	Redraw map (`Encode`)
	Move profiles highlight (`Select`)
	Move meteograms highlight (`Select`)
Panning/zooming map	Update map (`Navigate`)
Mouse over a cell	Show cell tooltip on the map (`Annotate`)
Click on a cell	Highlight cell on the map (`Select`)
	Redraw meteograms (`Encode`)
Mouse over a profile column	Show profile tooltip (`Annotate`)
Mouse over a meteogram	Show property value for the timestep (`Annotate`)

For the `visualization components` and `data services` we established an interface that should be implemented and "abstract" classes that could be used by the concrete implementation. For this implementation, both the `visualization components` and `data services` were developed in the same application as WISE (as opposed to being developed as separate applications). We chose this approach so the development of the VAApp would be clearer to the VAApp development team, with methods adapted to the context instead of a more generic one (for example, the map visualization component has a `selectCell` method instead of a generic `select` one).

All the implementation is done using TypeScript,[5] "a typed superset of JavaScript that compiles to plain JavaScript." This made development easier, with the possibility of defining interfaces to objects and classes, and also enabling the abstract class concept (one not native to JavaScript). Moreover, during development, this approach made debugging easier, with compatibility errors being noticed at compile time instead of run time.

Figure 6 shows a sample interaction log history visualization for the WISE system. We highlight the parameters with monospaced font in the `view navigation` and `user action` descriptions to make it easier for users to distinguish them from regular text. Another change was color-coding the `visualization action` tasks. Together with the "tag-like" appearance (text in a colored background) of the `visualization action` description, we believe this may aid users to establish the mapping with the task (the circle) and with the `visualization component` (the line in which the circle appears), whilst also working as a legend for the graph.

[5] http://www.typescriptlang.org/.



portal

19/Jan/2016 10:44:55 Navigated to WISE-SPL:portal

19/Jan/2016 10:44:55 ▸ Loaded page with forecast 2015-12-11 00:00:00, grid 9km Grid, property Temperature at timestep 2015-12-11T00:00:00.000Z.

19/Jan/2016 10:45:08 ▸ Changed timestep to 2015-12-11T14:00:00.000Z.

portal

19/Jan/2016 10:45:31 Navigated to WISE-SPL:portal

19/Jan/2016 10:45:31 ▸ Loaded page with forecast 2015-12-11 00:00:00, grid 9km Grid, property Temperature at timestep 2015-12-11T00:00:00.000Z.

19/Jan/2016 10:45:41 ▸ Changed forecast to 2015-12-10 00:00:00.

19/Jan/2016 10:45:57 ▸ Changed timestep to 2015-12-11T14:00:00.000Z.

19/Jan/2016 10:46:14 ▸ Selected cell -22.92x316.83.

19/Jan/2016 10:46:22 ▾ Selected cell -22.92x316.83.

19/Jan/2016 10:46:24 Select map

19/Jan/2016 10:46:24 Encode meteograms

Fig. 6. Sample history view excerpt for WISE interaction.

While developing this early implementation, the history visualization already proved itself useful for the development team, since it enabled visualizing the inner workings of the code. WISE's development team was able to detect some problems with the information flow, such as calls to the `data service` and drawing the `visualization components` in the wrong order or even when not yet necessary.

6 Final Remarks and Future Work

In this paper we provided a complete overview of our solution to record and present user interaction history. We started by introducing how our framework components work, continued to explain the log model that enables the communication between the VAApp and the history viewer, and discussed the main concepts and ideas of our history visualization system. We also presented our early concrete implementation of the solution using WISE as the origin VAApp, the results of such effort, and the unexpected benefit for the VAApp's development team.

The solution is yet in its early stages, needing more use cases and VAApps to strengthen the approach. We also need to evaluate the history visualization with users in order to better assess the visualization's efficacy, efficiency, and understandability.

We currently have plans to enhance the history visualization. We are aware of the scalability issue – when there are many columns, possibly with several VAApps, each with a multitude of view and visualization components. Besides common search, hide, and filter features, we plan to provide a more close-packed representation, for example by collapsing the lines and hiding the nodes, painting the lines themselves. Moreover, we plan to have some sort of mechanism to highlight common `visualization states`, as many different actions can lead to the same state. This could be used to hide sections of user actions (*e.g.*: collapse a sequence of actions that lead to an undo) and also to detect patterns in the user interaction.

Acknowledgement. The authors would like to thank CNPq for the financial support to their work (processes #309828/2015-5 and #453996/2014-0).

References

1. Aigner, W., Miksch, S., Müller, W., Schumann, H., Tominski, C.: Visualizing time-oriented data - a systematic view. Comput. Graph. **31**(3), 401–409 (2007). http://www.sciencedirect.com/science/article/pii/S0097849307000611
2. Brehmer, M., Munzner, T.: A multi-level typology of abstract visualization tasks. IEEE Trans. Vis. Comput. Graph. **19**(12), 2376–2385 (2013)
3. Green, T., Ribarsky, W., Fisher, B.: Visual analytics for complex concepts using a human cognition model. In: IEEE Symposium on Visual Analytics Science and Technology, VAST 2008, pp. 91–98, October 2008
4. Keim, D.A., Mansmann, F., Oelke, D., Ziegler, H.: Visual analytics: combining automated discovery with interactive visualizations. In: Boulicaut, J.-F., Berthold, M.R., Horváth, T. (eds.) DS 2008. LNCS (LNAI), vol. 5255, pp. 2–14. Springer, Heidelberg (2008)
5. Key, A., Howe, B., Perry, D., Aragon, C.: Vizdeck: self-organizing dashboards for visual analytics. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, pp. 681–684. ACM, New York (2012)
6. Kohlhammer, J., Keim, D., Pohl, M., Santucci, G., Andrienko, G.: Solving problems with visual analytics. Procedia Comput. Sci. **7**, 117–120 (2011). Proceedings of the 2nd European Future Technologies Conference and Exhibition 2011 (FET11). http://www.sciencedirect.com/science/article/pii/S1877050911007009
7. Mennis, J., Guo, D.: Spatial data mining and geographic knowledge discovery-an introduction. Comput. Environ. Urban Syst. **33**(6), 403–408 (2009). Spatial Data Mining–Methods and Applications. http://www.sciencedirect.com/science/article/pii/S0198971509000817
8. Oliveira, I., Segura, V., Nery, M., Mantripragada, K., Ramirez, J.P., Cerqueira, R.: WISE: A web environment for visualization and insights on weather data. In: WVIS - 5thWorkshop on Visual Analytics, Information Visualization and Scientific Visualization, SIBGRAPI 2014, pp. 4–7 (2014). http://bibliotecadigital.fgv.br/dspace/bitstream/handle/10438/11954/WVIS-SIBGRAPI-2014.pdf?sequence=1
9. Souza, C.S.D., Prates, R.O., Carey, T.: Missing and declining affordances: are these appropriate concepts? J. Braz. Comput. Soc. **7**, 26–34 (2000)